Python Key Notes

1. **Introduction Python**
2. **Variable**
3. **Data Type**
4. **Operator**
5. **Casting**
6. **Conditional Statement**
7. **Loops**
8. **Functions**
9. **Map Function**
10. **Filter Function**
11. **Reduce Function**
12. **OOPs Concepts**
13. **Exception Handling**
14. **File Handling**
15. **Python Collections**
16. **Practice Question of Python**

Introduction Python

Python is a widely used high-level, interpreted programming language. It was created by **Guido van Rossum in 1991** and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. Python is a programming language that lets you work quickly and integrate systems more efficiently.

Python is widely used language

- **Web Development**: Frameworks like Django, Flask.
- **Data Science and Analysis**: Libraries like Pandas, NumPy, Matplotlib.
- **Machine Learning and AI**: TensorFlow, PyTorch, Scikit-learn.
- **Automation and Scripting**: Automate repetitive tasks.
- **Game Development**: Libraries like Pygame.
- **Web Scraping**: Tools like BeautifulSoup, Scrapy.
- **Desktop Applications**: GUI frameworks like Tkinter, PyQt.
- **Scientific Computing**: SciPy, SymPy.
- **Internet of Things** (IoT): MicroPython, Raspberry Pi.
- **DevOps and Cloud**: Automation scripts and APIs.
- **Cybersecurity**: Penetration testing and ethical hacking tools.

Key Features of Python

- **Python is Easy to Learn and Use**: There is no prerequisite to start Python, since it is Ideal programming language for beginners.
- **High Level Language**: Python don't let you worry about low-level details, like memory management, hardware-level operations etc.
- **Python is Interpreted**: Code is executed line-by-line directly by interpreter, and no need for separate compilation. Which means –
    - You can run the same code across different platforms.
    - You can make the changes in code without restarting the program.
- **Dynamic Typed**: Python is a dynamic language, meaning there are no need to explicitly declare the data type of a variable. Type is checked during runtime, not at compile time.
- **Object Oriented**: Python supports object-oriented concepts like classes, inheritance, and polymorphism etc. OOPs empowers Python with modularity, reusability and easy to maintain code.
- **Extensive Library are Available**: Python has huge set of library and modules, which can make development lot easier and faster.
- **Open-Source with Huge community Support**: Along with opensource, Python is blessed with very large community contributing to its further development.
- **Cross Platform**: Same Python code can run on Windows, macOS and Linux, without any modification in code.
- **Good Career Opportunities**: Python is in high demand across industries like Software development, AI, finance, and cloud computing etc.

Limitation of Python

- **Performance Limitations**: Python is an interpreted language which makes it slower than compiled languages like C++ or Java.
- **Not preferred for Mobile Apps and Front End**: Python isn't preferred for Front End dev or mobile apps because of slower execution speeds and limited frameworks compared to JavaScript or Swift.
- **Memory Expensive**: Python consumes more memory because of its high-level nature and dynamic typing.
- **Lack of Strong Typing**: Dynamic typing makes Python easy to code, but it can lead to runtime errors that would be caught at compile time in statically typed languages like Java or C#.
- **Not Suitable for Game development**: Python lacks the high speed and low-level hardware control needed for game engines

Comment lines of code in python

```
# I am single line comment
# more hash then comment multiple line
```

Input/output

- **Input from the user**

- Use the `input()` function to take input from the user.
- Example –

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))  # Convert input to an integer
print(f"Hello, {name}. You are {age} years old.")
```

- **Output to the user**

- Use the `print()` function to display output to the user.
- Example –

```
print("Welcome to Python!")
print("The sum of 2 and 3 is:", 2 + 3)
```

Variable in Python

Variables are used to store data that can be referenced and manipulated during program execution. A variable is essentially a name that is assigned to a value

- No explicit type declaration needed (dynamically typed).
- Declared using = (assignment operator).
- **Rules for Naming Variables**
  - Variable names can only contain letters, digits and underscores (_).
  - A variable name cannot start with a digit.
  - Variable names are case-sensitive (myVar and myvar are different).
  - Avoid using Python keywords (e.g., if, else, for) as variable names.

Valid Example:

```
age = 21
_colour = "lilac"
total_score = 90
```

Invalid Example:

```
1name = "Error"  # Starts with a digit
class = 10       # 'class' is a reserved keyword
user-name = "Doe"  # Contains a hyphen
```

```
x = 10          # Integer
name = "Navin"  # String
pi = 3.14       # Float
```

References

- Variables in Python are references to objects in memory.
- The same object can have multiple references.

Example –

```
a = [1, 2, 3]
b = a  # Both 'a' and 'b' reference the same list
b.append(4)
print(a)  # Output: [1, 2, 3, 4]
```

Data Types

- Python provides various built-in data types:

Type	Example
`int`	`x = 10`
`float`	`pi = 3.14`
`str`	`name = "Navin"`
`bool`	`is_valid = True`
`list`	`fruits = ["apple", "banana"]`
`tuple`	`coords = (10, 20)`
`dict`	`person = {"name": "Navin"}`
`set`	`unique = {1, 2, 3}`
`NoneType`	`x = None`

- Use `type()` to check a variable's type:

```
print(type(10))  # Output: <class 'int'>
```

Operators

- **Arithmetic Operators**

Operator	Meaning	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus	`x % y`
`**`	Exponentiation	`x ** y`
`//`	Floor Division	`x // y`

- **Comparison Operators**

Operator	Meaning	Example
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`>`	Greater than	`x > y`
`<`	Less than	`x < y`
`>=`	Greater or equal	`x >= y`
`<=`	Less or equal	`x <= y`

- **Logical Operators**

Operator	Meaning	Example
`and`	Logical AND	`x > 5 and y < 10`
`or`	Logical OR	`x > 5 or y < 10`
`not`	Logical NOT	`not(x > 5)`

- **Assignment Operators**

Operator	Meaning	Example
`=`	Assign	`x = 5`
`+=`	Add and assign	`x += 5`
`-=`	Subtract and assign	`x -= 5`
`*=`	Multiply and assign	`x *= 5`

Casting

Casting is the process of converting a variable from one data type to another. Python provides built-in functions to explicitly cast data types

Function	Purpose	Example	Output
`int()`	Converts to integer	`int(3.7)`	`3`
`float()`	Converts to float	`float("10")`	`10.0`
`str()`	Converts to string	`str(123)`	`"123"`
`bool()`	Converts to boolean	`bool(0)`	`False`
`list()`	Converts to list	`list("abc")`	`['a', 'b', 'c']`
`tuple()`	Converts to tuple	`tuple([1, 2])`	`(1, 2)`
`set()`	Converts to set	`set([1, 1, 2])`	`{1, 2}`
`dict()`	Converts to dictionary	`dict([("a", 1)])`	`{'a': 1}`
`complex()`	Converts to complex number	`complex(3, 4)`	`(3+4j)`

- **Invalid conversions** (e.g., `int("abc")`) will raise errors.
- Use **`type()`** to verify types after conversion.

Conditional Statement

Python uses conditional statements to execute code based on conditions.

- Syntax:

```
if condition:
    # Code if True
elif condition2:
    # Code if condition2 is True
else:
    # Code if all conditions are False
```

- Key Types:
    - **if**: Executes if the condition is True.
    - **if-else**: Adds an alternate path if the condition is False.
    - **if-elif-else**: Tests multiple conditions.
    - **Nested if**: if inside another if.
    - **Ternary Operator**: One-line conditional:

```
result = "Yes" if condition else "No"
```

Example –

```
x = 10
if x > 5:
    print("Greater than 5")
elif x == 5:
    print("Equal to 5")
else:
    print("Less than 5")
```

Loops in Python

Loops are used to execute a block of code repeatedly until a condition is met.

- **for Loop**
- Iterates over a sequence (list, string, tuple, range, etc.).

Syntax -

```
for variable in sequence:
    # Code block
```

Example –

```
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

- **while Loop**
- Repeats as long as the condition is True.

Syntax –

```
while condition:
    # Code block
```

Example –

```
count = 0
while count < 5:
    print(count)
    count += 1  # Increment count
```

- **break Statement**
- Exits the loop prematurely.

Example –

```
for i in range(10):
    if i == 5:
        break
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

- **`continue` Statement**
- Skips the rest of the loop body for the current iteration and moves to the next iteration.

Example –

```
for i in range(5):
    if i == 3:
        continue
    print(i)  # Outputs: 0, 1, 2, 4
```

- **`else` with Loops**
- Executes after the loop finishes normally (not if exited by `break`).

Example –

```
for i in range(5):
    print(i)
else:
    print("Loop finished")  # Outputs after the loop
```

- **Nested Loops**
- A loop inside another loop.

Example –

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Functions

A function is a reusable block of code designed to perform a specific task

- **Defining a Function**
- Use the `def` keyword to create a function.

```
def function_name(parameters):
    # Code block
    return result
```

Example –

```
def greet(name):
    return f"Hello, {name}!"
```

- **Calling a Function**
- Invoke a function by its name with arguments.

```
print(greet("Navin"))  # Output: Hello, Navin!
```

Types of Functions

- **Built-in Functions**: Predefined, like print(), len().
- **User-defined Functions**: Custom functions created with def.

Function Parameters

- **Positional Parameters**: Pass arguments in order.
- **Default Parameters**: Provide default values.
- **Arbitrary Arguments**
    - `*args`: Pass multiple positional arguments as a tuple.

```
def add(*numbers):
    return sum(numbers)
print(add(1, 2, 3))  # Output: 6
```

    - `**kwargs`: Pass multiple keyword arguments as a dictionary.

```
def details(**info):
    return info
print(details(name="Navin", age=25))  # Output: {'name': 'Navin', 'age': 25}
```

Lambda (Anonymous) Functions

- Short, one-line functions using the `lambda` keyword.

Example –

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

Return Statement

- Ends a function and returns a value.

```
def square(x):
    return x * x
print(square(4))  # Output: 16
```

Map Functions

- The `map()` function applies a given function to each item of an iterable (e.g., list, tuple) and returns a new map object (an iterator).

Syntax –

```
map(function, iterable)
```

- **`function`**: The function to apply to each item in the iterable.
- **`iterable`**: The input sequence (e.g., list, tuple, set).

- **Using a Built-in Function**

```
numbers = [1, 2, 3, 4]
squared = map(pow, numbers, [2, 2, 2, 2])  # Raises each number to the power 2
print(list(squared))  # Output: [1, 4, 9, 16]
```

- **Using a User-Defined Function**

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4]
squared = map(square, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

- **Using Lambda Functions**

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

- **Multiple Iterables**

```
a = [1, 2, 3]
b = [4, 5, 6]
summed = map(lambda x, y: x + y, a, b)
print(list(summed))  # Output: [5, 7, 9]
```

When to Use map()

- For simple transformations of an iterable
- or operations that can be expressed as a single function or lambda

Filter Function

- The `filter()` function filters elements from an iterable based on a condition defined by a function. It returns an iterator with elements for which the function evaluates to `True`.

Example –

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
evens = filter(is_even, numbers)
print(list(evens))  # Output: [2, 4, 6]
```

Reduce Function

- The `reduce()` function in Python applies a function cumulatively to the items of an iterable, reducing it to a single value. It is part of the `functools` module.

Syntax –

```
from functools import reduce

reduce(function, iterable, initializer=None)
```

- `function`: A function that takes two arguments and returns a single value.
- `iterable`: The sequence to process.
- `initializer` *(optional)*: A starting value for the reduction.

Example –

```
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, numbers)
print(result)  # Output: 10
```

- Using an Initializer

```
numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, numbers, 10)  # Start with 10
print(result)  # Output: 20
```

- Finding the Maximum

```
numbers = [1, 5, 2, 8, 3]
result = reduce(lambda x, y: x if x > y else y, numbers)
print(result)  # Output: 8
```

OOPs Concept

Object-Oriented Programming is a paradigm where code is organized into "objects," which bundle **data** (attributes) and **functions** (methods) that operate on the data. This approach models real-world entities and their interactions.

- **Encapsulation**

- Encapsulation means bundling data (attributes) and methods (functions) that operate on the data into a single unit (class)
- Controls access to data using **public, private, and protected** attributes.

Example

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner       # Public attribute
        self.__balance = balance # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):  # Controlled access to private attribute
        return self.__balance

account = BankAccount("Navin", 1000)
print(account.get_balance())  # Output: 1000
```

- **Inheritance**

- Enables one class (child) to acquire properties and behaviors of another class (parent).
- Supports **code reuse** and the creation of hierarchical relationships.
- Inheritance allows a class (child) to inherit attributes and methods from another class (parent).

Example

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):  # Inheriting from Animal
    def speak(self):  # Method overriding
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Dog barks
```

- **Polymorphism**

- Allows methods in a class to have the same name but behave differently based on the object.
- **Method Overloading**: Achieved by default parameters.
- **Method Overriding**: Redefining a method in the child class.

Example

```
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):  # Overriding area method
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):  # Overriding area method
        return 3.14 * self.radius ** 2

# Example usage
shapes = [Rectangle(4, 5), Circle(3)]
for shape in shapes:
    print("Area:", shape.area())
```

- **Abstraction**

- Hides implementation details and only exposes essential features.
- Achieved using abstract base classes (via `abc` module).

Example

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Rectangle(Shape):
    def area(self):
        return self.width * self.height
```

Components of OOP

- **Classes**

- Blueprint for creating objects. Defines attributes and methods.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

- **Objects**

- Instances of a class.

```
person1 = Person("Navin", 25)
person1.greet()  # Output: Hello, my name is Navin and I am 25 years old.
```

- **Methods**

- Functions defined inside a class to represent behaviors of objects.

```
class Calculator:
    def add(self, a, b):
        return a + b
calc = Calculator()
print(calc.add(5, 3))  # Output: 8
```

- **Static Methods**

- Use `@staticmethod` decorator for methods that do not access instance or class variables.

```
class MyClass:
    @staticmethod
    def utility():
        print("Utility method")
MyClass.utility()
```

- **Constructor**

- Special method `__init__` initializes object attributes when an object is created.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

my_book = Book("1984", "George Orwell")
print(my_book.title)  # Output: 1984
```

- **Access Modifiers**

- `Public`: Accessible everywhere.
- `Protected` (_): Intended for internal use but accessible outside.
- `Private` (__): Not directly accessible outside the class.

- **Special Methods (Dunder Methods)**

Method	Purpose	Example
`__init__`	Constructor	Initializes attributes
`__str__`	String representation	`print(object)`
`__add__`	Overloads `+` operator	`a + b`
`__len__`	Returns length	`len(object)`

Exception Handling

- **Purpose**: To handle runtime errors and prevent program crashes.
- **Key Blocks**:

- **`try`**: Code that might raise an exception.
- **`except`**: Code to handle the exception.
- **`else`**: Executes if no exceptions occur.
- **`finally`**: Executes always, for cleanup.

Syntax –

```
try:
    # Risky code
except ExceptionType as e:
    # Handle the exception
else:
    # Execute if no exception
finally:
    # Cleanup code
```

Example –

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input!")
else:
    print(f"Result: {result}")
finally:
    print("Execution complete.")
```

File Handling and I/O

- **Opening a file:** Use the `open()` function to open a file. Modes

- `'r'` (read, default)
- `'w'` (write, overwrites the file)
- `'a'` (append)
- `'b'` (binary mode)
- `'+'` (read/write)

Example –

```
file = open("example.txt", "r")  # Open a file in read mode
content = file.read()            # Read file content
file.close()                     # Always close the file after use
```

- **Writing to a file**

```
with open("output.txt", "w") as file:  # Use "with" to auto-close the file
    file.write("Hello, File!\n")
    file.write("This is a new line.")
```

- **Appending to a file**

```
with open("output.txt", "a") as file:
    file.write("\nAppending another line.")
```

- **Reading from a file**

```
with open("example.txt", "r") as file:
    content = file.read()        # Read entire file content
    print(content)

with open("example.txt", "r") as file:
    lines = file.readlines()     # Read all lines into a list
    print(lines)

with open("example.txt", "r") as file:
    for line in file:            # Iterate over each line
        print(line.strip())      # Strip newline characters
```

- **Binary File Operations**

```
with open("image.png", "rb") as binary_file:
    data = binary_file.read()  # Read binary data

with open("copy.png", "wb") as binary_file:
    binary_file.write(data)    # Write binary data
```

- **JSON I/O**

```
import json

# Writing JSON
data = {"name": "Navin", "age": 28}
with open("data.json", "w") as file:
    json.dump(data, file)

# Reading JSON
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

- **CSV I/O**

```
import csv

# Writing CSV
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Navin", 28])

# Reading CSV
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

- **OS Operations (e.g., Checking/Deleting Files)**

```
import os

# Check if file exists
if os.path.exists("example.txt"):
    print("File exists.")
else:
    print("File does not exist.")

# Delete a file
os.remove("output.txt")
```

Python Collections

Python provides several built-in collection types to handle groups of data. These are flexible and powerful tools for various data manipulation tasks.

List

A list is an ordered, mutable collection of items. It can contain elements of different data types.

- Ordered
- Mutable
- Allows duplicate values

Common operations of list –

```
# Create a list
fruits = ["apple", "banana", "cherry"]

# Access elements
print(fruits[0])          # Output: apple

# Modify elements
fruits[1] = "blueberry"

# Add elements
fruits.append("date")     # Add to the end
fruits.insert(1, "pear")  # Insert at index 1

# Remove elements
fruits.remove("apple")    # Remove by value
popped = fruits.pop()     # Remove last element and return it

# Iterate
for fruit in fruits:
    print(fruit)

# List comprehensions
squared = [x**2 for x in range(5)]  # [0, 1, 4, 9, 16]
```

Tuple

A tuple is an ordered, immutable collection of items. Once created, it cannot be modified.

- Ordered
- Immutable
- Allows duplicate values

Common Operations of tuple –

```
# Create a tuple
numbers = (1, 2, 3)

# Access elements
print(numbers[0])         # Output: 1

# Unpacking
a, b, c = numbers

# Tuple methods
print(numbers.count(2))   # Count occurrences of 2
print(numbers.index(3))   # Find index of 3
```

Set

A set is an unordered collection of unique items.

- Unordered
- Mutable (elements can be added or removed)
- Does not allow duplicates

Common operations of set –

```
# Create a set
unique_numbers = {1, 2, 3, 4}

# Add elements
unique_numbers.add(5)

# Remove elements
unique_numbers.remove(3)    # Raises KeyError if not found
unique_numbers.discard(6)   # Does not raise an error

# Set operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}

print(set1.union(set2))     # {1, 2, 3, 4, 5}
print(set1.intersection(set2))  # {3}
print(set1.difference(set2))    # {1, 2}
```

Dictionary

A dictionary is an unordered collection of key-value pairs.

- Unordered (in Python 3.7+, insertion order is preserved)
- Keys must be unique and immutable
- Mutable

Common operations of dictionary –

```python
# Create a dictionary
person = {"name": "Navin", "age": 28}

# Access elements
print(person["name"])       # Output: Navin

# Add/Update elements
person["age"] = 29
person["city"] = "Delhi"

# Remove elements
del person["city"]
age = person.pop("age")     # Removes and returns the value

# Iterate
for key, value in person.items():
    print(key, value)

# Check for key
if "name" in person:
    print("Key exists")
```

Collections Module

- **Counter**: Counts occurrences of elements.

```
from collections import Counter
counts = Counter([1, 2, 2, 3, 3, 3])
print(counts)  # Counter({3: 3, 2: 2, 1: 1})
```

- **defaultdict**: A dictionary with a default value for missing keys.

```
from collections import defaultdict
d = defaultdict(int)
d["a"] += 1
print(d)  # defaultdict(<class 'int'>, {'a': 1})
```

- **deque**: A double-ended queue for fast appends and pops.

```
from collections import deque
dq = deque([1, 2, 3])
dq.append(4)
dq.appendleft(0)
dq.pop()
dq.popleft()
```

- **OrderedDict**: A dictionary that remembers insertion order (not needed in Python 3.7+).

```
from collections import OrderedDict
od = OrderedDict()
od["a"] = 1
od["b"] = 2
print(od)  # OrderedDict([('a', 1), ('b', 2)])
```

- **namedtuple**: Immutable, named fields for tuples.

```
from collections import namedtuple
Point = namedtuple("Point", ["x", "y"])
p = Point(10, 20)
print(p.x, p.y)
```

Choosing the Right Collection

Requirement	Collection
Maintain order	List, Tuple, OrderedDict
Unique elements	Set
Key-value pairs	Dictionary
Count occurrences	Counter
Fast insertion/removal from both ends	deque
Immutable collection	Tuple, namedtuple